# *Matthew Kramer – Assignment 2 Writeup*

The first step in developing this exploit is to run the target program – “getscore_heap.c” – using standard parameters. The program takes two arguments within the command line: a name and a social security number. Upon further examination of the source code, we find that a heap overflow exploit may be deployed since there are no checks in place to prevent arguments that exceed the capacity of the memory allocated for the string buffers. In particular, within “getscore_heap.c” – the malloc called in line 40 uses the ‘name’ parameter of the program as an argument for how much memory it should allocate.

To begin generating an exploit for this program, there are a couple of address that we need to find. The first of which is the GOT-table entry for the ‘free()’ function. To do this, we can simply call for an object dump of the getscore_heap executable that will display all of the functions used by the program and the addresses that they reside in.

```
[root@localhost heap]# objdump -R getscore_heap
getscore_heap:     file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ca4 R_386_GLOB_DAT    __gmon_start__
08049c5c R_386_JUMP_SLOT   perror
08049c60 R_386_JUMP_SLOT   system
08049c64 R_386_JUMP_SLOT   malloc
08049c68 R_386_JUMP_SLOT   time
08049c6c R_386_JUMP_SLOT   fgets
08049c70 R_386_JUMP_SLOT   strlen
08049c74 R_386_JUMP_SLOT   __libc_start_main
08049c78 R_386_JUMP_SLOT   strcat
08049c7c R_386_JUMP_SLOT   printf
08049c80 R_386_JUMP_SLOT   getuid
08049c84 R_386_JUMP_SLOT   ctime
08049c88 R_386_JUMP_SLOT   setreuid
08049c8c R_386_JUMP_SLOT   exit
08049c90 R_386_JUMP_SLOT   free
08049c94 R_386_JUMP_SLOT   fopen
08049c98 R_386_JUMP_SLOT   sprintf
08049c9c R_386_JUMP_SLOT   geteuid
08049ca0 R_386_JUMP_SLOT   strcpy
```

Here we can see the free function has an offset of 0x8049c90. Next, we will need to determine the base address of the buffer we are targeting. By looking at the source code for getscore_heap, we can see that this buffer is allocated on line 40. In order to find the address of this buffer, we will need to stop the program once it has allocated it and examine the stack. To do this, we will compile the program with the debugging flag and open the executable in the GNU Debugger.

```
[root@localhost heap]# gcc -g -o getscore_heap getscore_heap.c
[root@localhost heap]# gdb getscore_heap
```

We’ll begin by setting the arguments of the program; otherwise, the program would never begin running.

```
(gdb) set args “aaa” “123456789”
```

With the arguments set, we must now set a breakpoint after the buffer is allocated, as we mentioned before. Otherwise, the program will execute as it did outside of the debugger, giving us no time to examine the stack as instructions are executed. We found previously that the buffer is allocated on line 40 of the program, however this line is followed by the conditional block of an if-statement. To be safe, we will set a breakpoint after this statement, ensuring that the program will break where we want it to.

```
(gdb) break 44
Breakpoint 1 at 0x80487bc: file getscore_heap.c, line 44.
```

With the breakpoint set, we can now run the program within the debugger.

```
(gdb) run
     Starting program: /root/heap/getscore_heap "aaa" "123456789"
     Breakpoint 1, main (argc=3, argv=0xbffffad4) at getscore_heap.c:45
     warning: Source file is more recent than executable.

     45        if ((score = (char *)malloc(10)) == NULL){
```

Success! We have paused the program's execution just after the buffer has been allocated. With the program now in this state, we can examine the code further to determine the location of our buffer. To do this, we can execute the print command with the name of the buffer in question.

```
(gdb) print matching_pattern
$1 = 0x8049e28 ""
```

Here we can see that the buffer matching_pattern has a base address of 0x8049e28. This is the second and final piece of information that we need to generate the exploit. The final exploit will be composed of two strings, as the program requires two arguments and they are both required due to the handling of the heap structure. Refer to the source code of this exploit to see how the exploit is generated. I have left comments on every bit of code to explain what is going on and how the final exploit is formed.